新人技術者のための

ロジカル・シンキング入門 第7回

「ひたすら流すだけ」にさようなら

冴木 元

LOGICAL THINKING

「テスト・ケースはテストを実施する前に作成しておくもの」という当たり前のことを，実際の開発現場では実行できていないと感じている設計者は少なくないだろう．今回は，テスト・ケースを漏れなく，かつ効率良く作成するための考え方を説明する．開発するシステム（モジュール）に応じて，入力パラメータや入力データを考えて場合分けすることで，テスト・ケースを充実させていく．（編集部）

Gさんは，今月から新しい開発チームに所属することになった，入社2年目の若手のシステム・エンジニアです．

既存のプログラムのテスト工程から参加するようになったGさんに，リーダのPさんは「仕様書を見ながら担当するモジュールのテスト項目を考えてほしい」と伝えました．ところが，この当たり前の要求にGさんは思わず面食らってしまいました．「ん，なんだなんだ？ テスト項目って本当に考えるものなのか…．そんなのは新人研修用の建前だとばかり思っていたぞ…」．

● 納期前日に項目を作成，なんてことをしていませんか？

それもそのはず，Gさんがこの間までいた開発チームでは，テスト項目を用意して動作を確認することはほとんどなかったからです．

そのチームでは，開発スケジュールがあまりにも短期間であったため，コーディングとデバッグに多くの時間を費やしていました．結局，納期を大幅にずれこんで，やっとのことで顧客にシステムを提供したため，テスト項目を用意してバグを見つける暇などなかったのです．実運用開始までの間，本番用のデータをひたすら流し込み，システムがおかしな動作をすればその都度直していたのでした．そのため仕様が明確でない部分も多くあり，目の前で起こった事象がバグなのか仕様なのかをめぐって，深夜にもめごとになったことも何度かありました．

結局，「テスト項目」として納品したドキュメントは，納品日の前日に全員で急きょまとめ上げ，テスト日を工程上問題なさそうな日付に記したものだったのです．

Gさんはおずおずと Pさんに尋ねました．「あの～，参考にしたいので，このチームでこれまで使っていたようなドキュメントを見せていただけませんか」．すると，リーダのPさんは**表1**を見せ，手始めにテスト項目をツリー上に分類してみるように伝えました．「項目分けを表にして整理することで，過不足なくテスト項目が整理できるから，みんなそうしている」ということです．「なるほど．これな

表1
ツリー構造によるテスト・ケースの分類
ツリーを使ってテスト・ケースを分けるのはオーソドックスなやり方．分類を大きい方から小さい方へと分けていき，テストの漏れと重複がないようにしていく．問題はテスト・ケースの分け方である．

			テスト・データ	期待する結果
1 動画機能ブロック・テスト	1.1 通常処理	1.1.1 Aモード	Aモード入力データ	正常に描画すること
		1.1.2 Bモード	Bモード入力データ	正常に描画すること
	1.2 エラー処理	1.2.1 CRCエラー時	エラー・データ	エラー保護すること

KeyWord テスト，モジュール，テスト・ケース，全数テスト，組み合わせ回路，順序回路，ステート・マシン，入力パターン

ら取りあえず何か思い付くかもしれない」．ほっとしたGさんは，与えられた時間，じっくりとテスト項目を考えてみることにしました．

仮に，読者のみなさんがリーダのPさんだとしたら，Gさんが提出してくるドキュメントにどのようなことを期待するでしょうか．

● ロジカル・ツリーの分け方

Pさんが指示したようなテスト・ケースの分類はオーソドックスな方法であり，それ自体は至極まっとうなものです．とはいえ，実際にテスト・ケースを分類するとなると，経験の少ない人であればいろいろと悩むこともあるでしょう．たとえベテランと呼ばれているような人でも，初めて接するタイプのシステムであれば，なにかと工夫しなければならないことが生じることに気づくと思います．

ここでは，組み込み開発で必要となりそうなテスト・ケースを，次の二つの点に焦点を当て，なるべく一般論（アプリケーションに依存しない）で解説します．

- モジュール単体のテスト・ケースの考え方
- 単体テストと結合テストのすみ分け

● モジュール単体のテスト・ケースを考える

モジュール単体のテスト・ケースを考える場合，誰でも思い付きそうなことは，「モジュールに与えられたパラメータの組み合わせを考え，それらを一通りテストする」というものでしょう．**表1**のようなツリーを使った分類でも，パラメータの種類と個別のパラメータを分けてテスト・ケースを充実させていくのが普通のアプローチの仕方だと思います．実際，一般に普及しているソフトウェア工学ですと，このような考え方にたって「複合条件網羅」といったテスト・ケースの設計条件が整理されているようです．読者の中には情報処理試験の受験のためにそうした知識を覚えた方もいるでしょう．

しかしこの考え方には，見落とされている点があると筆者は考えます．それは，そのモジュールに「ステート」があるかないかでテスト・ケースに対する考え方が変わってくるということです．

ステートの有無というのは，要するにテスト対象のモジュールの中にスタティック・データが存在するかどうかです．スタティック・データとは，モジュール処理が終わっても次の処理まで値を保持するデータのことです．C言語であれば，外部変数を使うとか，`malloc`を使って実装するデータです（連載第2回，本誌2006年6月号，pp.60-66を参照）．このようなステートが存在するモジュールの場合，単純に入力パラメータの組み合わせを網羅してもテスト・ケースに漏れが生じます．

● ステートがなく，自動化できれば全数テストがお勧め

まず，ステートがないモジュールから考えることにしましょう．ここでは，このようなモジュールをハードウェア回路に倣って，「組み合わせ回路」と呼ぶことにします．

図1のようなAND回路を例にとって考えてみましょう．AND回路とは，**図1(b)**の真理値表のような動作を行う回路のことです（ハードウェア技術者の方にとっては当たり前過ぎて今更解説の必要などないかもしれないが…）．**図1(b)**を見ると，4通りの入力の組み合わせに対して，Yの期待値が与えられていることが分かります．入力信号の状態としては‘0’または‘1’しかないため，AとBの2種類で2^2＝4通りの入力パターンが考えられるというわけです．

このように，組み合わせ回路については，起こり得るすべての入力パターンを拾い上げれば全数テストになります．回路がステートを持たないため，入力に対して出力が一意に決まるということがその理由です．

この考え方によると，どんなに複雑な回路（あるいはモ

（a）回路記号

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

（b）真理値表

図1
AND回路の例
組み合わせ回路の場合，入力パターンをすべて掛け算すると全数テストになる．組み合わせ回路はステートを持たないからである．

図2
全数テストとその限界
全数テストを実施するのが確実だが，テスト・ケースが膨大になることもある．やみくもにテスト・ケースを増やせばよいというものでもないが，テストの実施を自動化して効率化を図れるのであれば頑張って全数テストを行うことも大事．

ジュール）であっても，それが組み合わせ回路である限り，入力パターンをすべて拾い上げれば全数テストになります．ハードウェアの観点からいうと「テストベンチ」，ソフトウェアの立場からいえば「テスト用メイン」を作って，テスト・ベクタ（テスト用の期待値データ）との比較でテストができるような場合，多少パターンが多くても全数テストを実施するのが確実といえそうです．例えば**図2**に示すように，入力信号がA～Eの五つで，それぞれ8個の入力パターンがある回路を考えると，$8^5 = 32768$通りのテスト・ケースがあることになります．

テスト・ケースの数だけを考えると，「ちょっとやり過ぎ」といえなくもありません．しかし，テストベンチを組んでテストを自動化してしまえるのであれば，労をいとわずに全数テストを実施してしまう方が後々楽ともいえます．なぜなら，テストが一度自動化されてしまえば，再テストの実施は短時間で実施できるようになるからです．

● ソフトでは限界値テストで絞り込みが必要なケースも

一方，ソフトウェアの場合，そうもいっていられないケースが起こり得ます．テスト用関数でテストを自動化できない場合です．例えば**図2**のような多数の入力パターンが存在するにもかかわらず，それらをすべて操作画面からユーザが手入力しないと出力が得られないようなシステム（あるいはモジュール）がそれにあたります．また，期待値と出力値のファイル比較で合否の判定ができない場合も自動化できません．例えば，目視しないと結果の合否を判定できないような場合などがこれにあたります．

このような場合は**表2**に示すように，各入力信号に限界値（表の濃い灰色の部分）を振り分けることによって，テスト・ケースの絞り込みを図るべきでしょう．自動化でテストを効率化できない場合にやみくもに全数テストにこだわるのは，かえってテストの質を下げてしまいかねません．なぜなら人間は，自分の作ったモジュールをテストする場合は特に「正しく動いてほしい」という思い込みがあるため，ついついOKと判定してしまいがちだからです．そのため，やみくもに数にこだわったテストは，繰り返し行えば行うほどその質を下げてしまうことになります．

● ステートありでは入力を網羅しても全数テストではない

次に，内部にステートを持つモジュールについて考えてみることにしましょう．ここでもハードウェア回路に倣って，このようなモジュールを「順序回路」と呼ぶことにします．順序回路とは，ハードウェアの観点からいうとフリップフロップを持つ回路です．このような回路はハードウェアであれソフトウェアであれ，内部にステートを持ちます．順序回路の特徴は，次の通りです．

①入力とステートの組み合わせで出力が決まる
②ステートには初期値が必要
③リセットが必要

これらの特徴を**図3**に示すTフリップフロップを例にとって考えてみます．

まず特徴①ですが，Tフリップフロップは，端子Tに‘1’が入るたびにQのステート（状態）が‘0’→‘1’→‘0’→‘1’と反転する（トグルする）回路のことです．入力が同じ（例えばTが‘1’）でも，‘0’と‘1’が交互に出力されることから，出力は入力だけでは決まらないことが分かります．出力の期待値は，あくまで入力とステートの組み合わせから決まります．

ステートを持っているので，ステートの値には初期値が必要であることが分かります（特徴②）．そうでなければ回路の中に不定値を持つ部分があることになってしまうからです．そこから，ステートを初期値に設定するリセット処

表2 限界値によるテスト・ケースの絞り込み
自動化が困難な場合などは限界値を適宜振り分け，テスト・ケースを絞り込むことも大事．パソコンのOSを介して手入力が必要なシステムなどの場合，自動化はそう簡単ではない．テスト・ケースの数のみに固執するとテストそのものがいいかげんになり，かえって品質低下を招くこともある．

ケース＼入力信号	A	B	C	D	E
1	7	0	2	4	6
2	1	7	0	3	5
3	2	4	7	0	6
4	1	3	5	7	0
5	0	2	4	6	7

⇩

32768通りのテスト・ケースから絞り込み

R	T	Q	Q ’	Q（次の状態）	Q ’（次の状態）	
0	―	―	―	0	1	初期化
1	0	0	1	0	1	変化せず
1	0	1	0	1	0	
1	1	0	1	1	0	反転
1	1	1	0	0	1	

（a）回路記号　（b）真理値表

図3 Tフリップフロップ
初期値が存在し，QとQ ’のステート（状態）が出力を決める．初期化時と通常動作時で表の組み立てが違うことに注目．

図4
順序回路の動作
ステートを持つ場合，スタティック変数の値と入力の組み合わせで出力が決まる．スタティック変数を毎回設定してテストできれば全数テストも可能．そのようなスタティック変数にモジュールの外から初期値以外の値が設定できるようにするには設計に工夫が必要．

図5　ステートを外から設定してテストする方法
モジュールの外からステートが設定できるようになっていれば，入力データやパラメータ，ステートを与えることで，出力とステートが期待値と一致するかを確認できる．設計やコーディングの際はテストのしやすさも考慮に入れること．

理が必要なことが分かります（特徴③）．これを実装の面から見ると，「ハードウェアではリセット信号が必要」ということになります．ソフトウェアであれば，モジュールにはリセット用の関数が必要ということになります（**図4**）．

以上のことから，順序回路の場合は入力パターンをすべて網羅しても全数テストになるわけではないということが分かります．

● ソフトウェアは比較的容易に全数テストできることも

このことは，時としてテストを困難なものにしてしまいます．なぜなら，モジュールの外からステートを直接設定できるような作りになっているとは限らないからです．これはハードウェア，ソフトウェアの両方の場合で共通していえます．

ハードウェアの場合，テスト回路を組んで，テスト用にステートを設定できるようにする手段もあるでしょう．しかし，回路の中にあるすべてのフリップフロップを望みのステートにするようにテスト回路を組むのは非現実的であるように思えます．順序回路の場合，組み合わせ回路とは別の事情から，テスト・ケースの絞り込みが必要となってしまいます．

しかしソフトウェアの場合，ステートの設定がモジュールの外からできるようになっていれば，テストは容易になります（**図5**）．本連載の第2回（本誌2006年6月号，pp.60-66）で示したように，ステート，すなわちスタティック・データを構造体からたどれるように実装した場合を考えてみます．この場合は，テスト対象となるモジュールの外から自由にスタティック・データの値を設定できるので，入力とステートのパターンをすべて網羅することもできます．

要するに，入力信号と同じように手軽にステートを設定できる場合，組み合わせ回路と同じ考え方で全数テストを行えるのです．

逆にステートの中身が関数の外から設定できない場合は，全数テストの実施が難しくなります．

連載第5回（本誌2006年10月号，pp.132-138）では，「設計，テスト，コーディングの各工程を調和させなければならない」と述べましたが，ここで示した例でも分かるように，設計やコーディングを行う段階でテストのしやすさまで考えることは重要です．

入力イベントとステートによって処理（アクション）と次のステートが定まる，いわゆる「ステート・マシン」は，このような全数テストが望ましい回路（モジュール）といえるでしょう．

要するに，入力とステートの組み合わせを状態遷移表にして，次の動きが一意に定まるような場合は，全数テストが有効，もしくは必要であるといえます．ステート・マシンの場合，状態遷移表の升目の数だけテスト・ケースを用意すれば全数テストとなります．ですから，テストのたびに升目にチェックを付けていき，表のすべてがチェック印で埋め尽くされたところでテストが完了します．

● 意外と難しい「単体テスト」，「結合テスト」の区別

次に，「単体テスト」と「結合テスト」という分類の仕方について考えてみることにしましょう．冒頭の例でいうと，

```
int GetLength(){
    return MAX_LENGTH;
}
```

リスト1　単体テストの実施が無意味な関数の例
見た目で内容が確認できるような小さな関数にも単体テストを実施するのはナンセンス．バグの原因として怖いのは，いかなる場合も正しいタイミングで必ず使われているかどうかであり，そのためのテスト項目がむしろ必要．

図6　単体テストは関数ではなく機能ブロックに対して行う
関数単位ではなく，機能ごとに動作確認を行うのが単体テスト．機能間のインターフェースを中心にブロックをつなげたときの動作確認を行うのが結合テスト．こう考えれば，単体テストと結合テストを併用することでバグを効果的に取り除ける．「単体＝関数」という考え方にとらわれるとうまくいかない．

Gさんが担当しているモジュールの単体テストを終えたら，次の工程で全体の結合テストを実施することになります．単体テストから結合テストへという進め方は，一つ一つ部品を確実に組み立てる立場からすれば当たり前のことのように思えます．しかし，実際の開発ではこの「単体」と「結合」の区別はそれほど容易ではありません．

開発者の中には，「単体テストとは関数一つ一つに対して行うもの．それらの確認が終わって初めて結合テストができる」とやや教条主義的に考えている人もいます．また，結合テストはビッグバン・テスト（単体が終わったら全体を一気につなげる）以外考えず，段階的に結合していくことにはあまり関心のない人もいます．このような考え方が行き過ぎると，いろいろな悲劇や喜劇を生む原因にもなります．

● 単体テストは関数単位とは限らない

いわゆるオブジェクト指向を採用したシステムでは，**リスト1**のような小さな関数にお目にかかることがよくあります．**リスト1**は，例えば処理フレームの大きさを知るためにこのようなインターフェースを用意したのだと考えられます．

このように関数化しておくと，復帰値が単なるdefine文による固定値ではなく，将来の仕様変更で何らかの計算を行うように変わったとしても，ほかの部分のソース・コードはこの関数を呼ぶ形のままで変わることはありません．いわゆる「カプセル化」と呼ばれる考え方です．

このような小さな関数の場合，関数単体の動作を確認するという行為はナンセンスです．なぜなら，関数が仕様通り作られているかいないかは，ソース・コードを見れば十分把握できるからです．小さな関数が多く集まったシステムの場合，バグの原因となるのはむしろ関数のコール・シーケンス，すなわち呼ばれるべきタイミングで呼ばれるかどうかということにつきます．**リスト1**のような関数の場合，この関数が呼ばれずにバッファ・サイズが不定値のままアロケートされてしまうようなことが例として挙げられます．こうしたバグは，関数単体をいくらテストしてもチェックすることはできません．

テスト項目とはすなわち，このようなバグを効果的に発見できるケースの組み合わせにほかならないのです．

● 単体テストは機能ブロック単位で

連載第3回（本誌2006年8月号，pp.102-108）でも解説しましたが，筆者は設計の単位は機能ブロックで分けていくべきだと考えています（**図6**）．ですから当然，「単体テスト」と呼ばれるテスト・ケースも，この機能ブロックが単位となります．この機能ブロックごとに正常系，異常系を含めてあらゆる動作パターンを検証し，バグがないかどうかを確認するようなテスト・ケースを組み立てることが重要となるのです．

例えば，**リスト1**に示した不定値によるメモリ・アロケートの例で考えてみます．このようなバグは，想定しなかった例外的なパターンで生じることが少なくありません．従って，機能ブロックの異常系のテスト項目を充実させてバグの混入を防ぐ，という対策を採ることができるでしょう．正常な動作のケースでは現れにくいバグだからです．

もちろん，あまりに想定外のパターンが生じることがないように作っておくことは，設計の段階から重要となります．異常ケースについてすべてのパターンを考えることはなかなか難しく，また時間的な制約も無視できないからです．

● 結合テストは一段階とは限らない

結合テストに話を移しましょう．機能ブロックを単位とする単体テストの考え方から，結合テストは機能ブロックを組み合わせて動作させたときの確認ということになります．ここでは機能ブロック間のインターフェースが特に重

図7　結合テストの段階的実施

図6のようなブロック構成だと，結合テストは②～⑤のフェーズが考えられる．システムが安定しないうちは②～④を経て⑤を実施し，安定してきたら⑤のみに絞ってテスト・ケースを充実させていく，といった工夫が必要．

要となります．**図6**の例でいうと，全体の入出力は①と②のデータだけですが，結合テストでは図の長円で囲った機能ブロック間のデータが正しくやりとりされているか，という確認も重要となります．機能ブロック間の連絡がうまくいっているかどうかを確認するのが結合テストのポイントだからです．

また，初めて作り上げた機能の場合，その機能ブロックがすぐに動くとは限りません．そのような場合，いきなりA，B，Cの三つのブロックを結合させてテストするのではなく，まず，AとB，BとC，AとCの組み合わせでテストする項目を立てることも必要だと思います．これら一つ一つの組み合わせで問題がないことを確認してから，全体を結合させてテストするということです（**図7**）．

もちろん，開発が進んで各機能ブロックが安定してくれば，このような部分の組み合わせの結合テストは重要性が低くなってくると思います．そのような場合，A，B，Cすべてを組み合わせたテスト・ケースに重点を移し，より派生的な機能や異常系の機能のテスト・ケースを充実させていくという進め方になるでしょう．

● テスト・ケースを「使い捨て」にしない

以上，テスト・ケースを組み立てる際に注意すべき点について解説しました．テスト・ケースは，基本的にさまざまな入力パラメータや入力データについていろいろと場合分けすることで充実させていくことができます．また，テスト・ケースは基本的にはテストする前に用意するものです．

しかし，テスト・ケースを見直すためには忘れてはならない重要な工程があります．それは，モジュールをリリースした後にバグが見つかったときです．十分にテストしたと思ったのにバグが見つかった場合に，テスト・ケースの見直しが必要になります．

もちろんバグは「作る」工程におけるミスが直接の原因なので，設計またはコーディングの工程におけるミスのせい，ということになるでしょう．これは否定のしようがないことです．しかし，同時にバグは「検証する」工程のミスでもあります．ほとんどの場合，リリース後のバグはテスト・ケースの漏れが原因で発生してしまうのです．

バグを発生させた後，「どのようなテストを実施すれば防げたか」を考えることは重要です．テスト・ケースを改善していくことは，開発対象となるモジュールだけでなく，後に行う同じような開発に対しても製品の品質向上に役立ちます．筆者の経験からいっても，重要なテスト・ケースが，「もともとは何らかのバグが発生してその予防策を考えたときに思いついたものだった」という例はいくつもあります．

テスト・ケースは1回の開発で使い捨てにされるものではありません．組み込みシステムにおいては，開発対象のハードウェア・アーキテクチャが変わればモジュールそのものは開発し直すことになります．しかし，最初の開発で使ったテスト・ケースは，多くの場合，次の開発でも財産になるのです．ハードウェア・アーキテクチャの改版が必要となる組み込みシステム開発では，このことは見逃せません．テスト・ケースは使い捨てにするべきではないのです[注1]．

「失敗から学ぶ」というのはあまりに使い古された言い方ですが，優れたテスト・ケースを考えるためには見落とすことのできない考え方です．

注1：ソフトウェアのテストに関する古典的な文献にもこのような指摘が見られる．例えば，参考文献(1)のp.17に記されている．

参考・引用*文献

(1) Glenford J. Myers；ソフトウェア・テストの技法，近代科学社，1980年3月．

さえき・はじめ

<筆者プロフィール>

冴木元．システム・エンジニア．昼休みに新聞を読むことが多いが，面白いと思うのは日本語で読める海外の新聞．最近領土問題で対立した某国の報道などは国内の新聞よりはるかに解説が詳しく専門的で，いろいろと考えさせられた．「相手の言い分を聞く」というのはいかなる紛争解決にも必要なようだ．